# SMoL

Standard Model of Languages

# SMoL

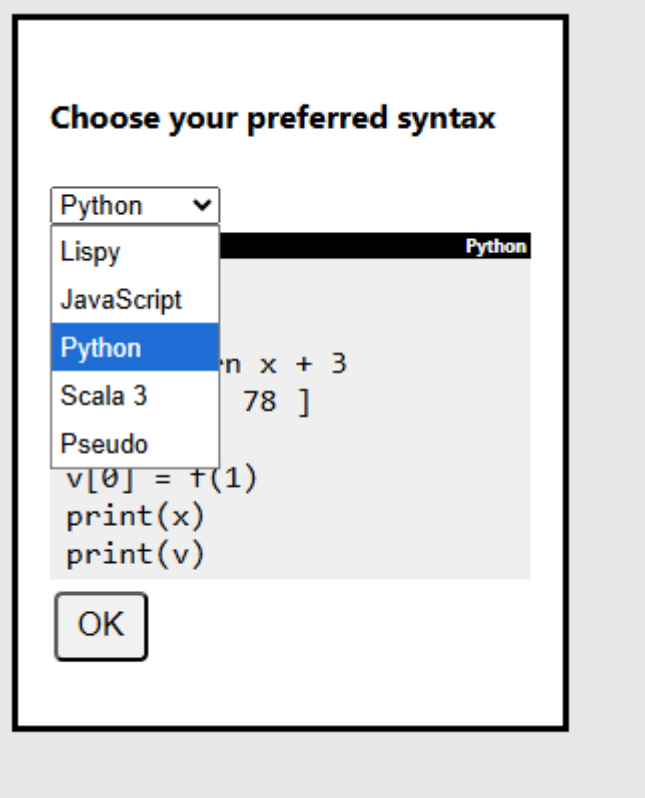
A tutoring system for teaching the semantics of programming languages (i.e., how programming

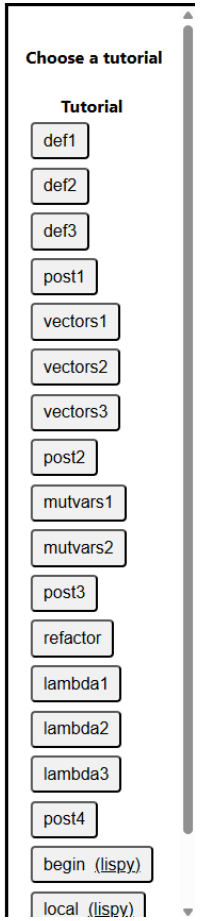
SMoL cannot save the state. That means you should finish every tutorial you start; you can’t pause and resume it later

# SMoL

- The SMoL Tutor breaks its content into *tutorial modules.*
- Each module contains multiple tasks/questions designed to teach and assess understanding of that concept
- Example: **scope Focus:** Variable and function definitions + lexical scope

# SMoL select the syntax



# Useful concepts to review

(Task 1 of 12) This series of tutorials revolve around a central idea, **SMoL**, the **S**tandard **M**odel **o**f **L**anguages. This is the embodiment of the computational core of many of our widely-used programming languages, from C# and Java to JavaScript, and Python to Scala and Racket. All these languages (and many others), to a large extent, have a common computational core:

- lexical scope
- nested scope
- eager evaluation
- sequential evaluation (per "thread")
- mutable first-order variables
- mutable first-class structures (objects, arrays, etc.)
- higher-order functions that close over bindings
- automated memory management (e.g., garbage collection)

Over the course of the SMoL Tutor, you will become familiar with these language features through reading and evaluating programs. What goes in SMoL is, of course, a judgment call: when a feature isn't present across a large number of diverse languages (like static types), or shows too much variation between languages that have it (like objects), we do not include that in the *standard* model. But it is not a value judgment: the standard model is about what languages *are*, rather than what languages *should be*.

# Lexical Scope

The scope of a variable is determined by the **physical structure of the program text**, not by how the program runs.

Python uses the **LEGB rule**:

- **Local**
- **Enclosing**
- **Global**
- **Built-in**

Python does *not* search “who called me.”
It searches “where am I written.”

```
x = 10

def f():
    print(x)

f()

x is not passed as a parameter

Python looks outward from f’s body

It finds x in the global scope
```

# Variable definitions

(Task 3 of 12) In this tutorial, we will learn about **definitions**.

The following program illustrates **variable definitions**.

Python
```
x = 1
y = 2
print(x)
print(y)
print(x + y)
```

Pseudo
```
let x = 1
let y = 2
print(x)
print(y)
print(x + y)
```

In this program, the first variable definition **binds** `x` to `1`, and the second variable definition binds `y` to `2`.

This program produces three values: the value of `x`, the value of `y`, and the value of `x + y`. These values are `1`, `2`, and `3`, respectively. In this Tutor, we will write the result of running this program on a single li

```
1 2 3
```

rather than

```
1
2
3
```

(Task 4 of 12) What is the result of running this program?

Python
```
x = 1
y = x + 2
print(x)
print(y)
```

JavaScript
```
let x = 1;
let y = x + 2;
console.log(x);
console.log(y);
```

```
1 3
```

You predicted the output correctly 🎉 🎉 🎉

The first definition binds `x` to the value `1`. The second definition binds `y` to the value of `x + 2`, which is `3`. So, the result of this program is `1 3`.

Click here to run this program in the Stacker.

# Function definitions

(Task 5 of 12) The following program illustrates **function definitions**.

Python

```
def f(x, y):
    return (x + y) / 2
print(f(2 * 3, 4))
```

Scala 3

```
def f(x : Int, y : Int) =
  (x + y) / 2
println(f(2 * 3, 4))
```

This program produces `5`. It defines a function named `f`, which has two **formal parameters**, `x` and `y`, and **calls** the function with the **actual parameters** `6` and `4`.

(Task 6 of 12) What is the result of running this program?

Python

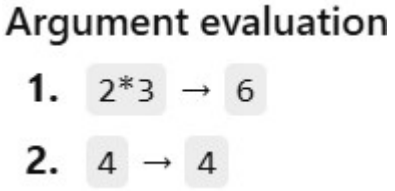
```
def f(x, y, z):
    return x + (y + z)
print(f(2, 1, 3))
```

Pseudo

```
fun f(x, y, z):
  return x + (y + z)
end
print(f(2, 1, 3))
```

- ○ `1`
- ○ `2`
- ○ `3`
- ○ `6`
- ○ `error`
- ◉ Other

# Comments




(Task 7 of 12) Function definitions can contain definitions, for example

Python

```
def f(x, y):
    n = x + y
    return n / 2
print(f(2 * 3, 4))
```

Pseudo

```
fun f(x, y):
  let n = x + y
  return n / 2
end
print(f(2 * 3, 4))
```

The name f is bound in the current (global) scope

A **new local scope** (stack frame) is created for this call:

x → 6
y → 4
local bindings

- eager evaluation: f(2*3, 4), 2*3 is evaluated **before** entering f**,** f never sees the expression 2*3, only the value 6
- lexical scope: n is defined **inside f, f** cannot be accessed outside the function print(n)
- nested definitions (local bindings): n is a local definition inside a function, it creates a local binding
- sequential evaluation: Inside f, statements execute top-to-bottom, define n, and computes and return expression using n.
- first-order mutation (binding): n is not a value passed around. It is a **local binding.**

# Blocks

(Task 9 of 12) We have two kinds of places where a definition might happen: the top-level **block** and function bodies (which are also **blocks**). A block is a sequence of definitions and expressions.

Blocks form a tree-like structure in a program. For example, we have four blocks in the following program:

Python

```
n = 42
def f(x):
    y = 1
    return x + y
def g():
    def h(m):
        return 2 * m
    return f(h(3))
print(g())
```

Lispy

```
(defvar n 42)

(deffun (f x)
  (defvar y 1)
  (+ x y))

(deffun (g)
  (deffun (h m)
    (* 2 m))
  (f (h 3)))

(g)
```

The blocks are:

- the top-level block, where the definitions of `n`, `f`, and `g` appear
- the body of `f`, where the definition of `y` appears, which is a sub-block of the top-level block
- the body of `g`, where the definition of `h` appears, which is also a sub-block of the top-level block, and
- the body of `h`, where no local definition appears, which is a sub-block of the body of `g`

# More about definitions

(Task 2 of 15) What is the result of running this program?

Python

```
x = 1
def f():
    return x
def g():
    x = 2
    return f()
print(g())
```

Lispy

```
(defvar x 1)
(deffun (f)
  x)
(deffun (g)
  (defvar x 2)
  (f))

(g)
```

```
1
```

You predicted the output correctly 🎉 🎉 🎉

`g()` evaluates to `f()`, which evaluates to `1` because `f` and `x` are both defined in the same block, and `f` does *not* get the `x` defined inside `g`.

Click here to run this program in the Stacker.

# What is the result of this program ?

(Task 3 of 15) What is the result of running this program?

Python

```
def f(x):
    y = 1
    return x + y
print(f(2) + y)
```

Scala 3

```
def f(x : Int) =
  val y = 1
  x + y
println(f(2) + y)
```

- ○ 1
- ○ 2
- ○ 3
- ○ 4
- ○ error
- ◉ Other

# What is the result of this program

(Task 4 of 15) What is the result of running this program?

Python

```
x = 1
def f():
    y = 2
    def g():
        return x + y
    return g()
print(f())
```

Scala 3

```
val x = 1
def f =
  val y = 2
  def g =
    x + y
  g
println(f)
```

# What is the result of this program

(Task 6 of 15) What is the result of running this program?

Python

```
x = 1
def f(y):
    def g():
        z = 2
        return x + y + z
    return g()
print(f(3) + 4)
```

Scala 3

```
val x = 1
def f(y : Int) =
  def g =
    val z = 2
    x + y + z
  g
println(f(3) + 4)
```

# What is the result of this program

(Task 6 of 15) What is the result of running this program?

Python

```
x = 1
def f(y):
    def g():
        z = 2
        return x + y + z
    return g()
print(f(3) + 4)
```

Scala 3

```
val x = 1
def f(y : Int) =
  def g =
    val z = 2
    x + y + z
  g
println(f(3) + 4)
```

# Concept

(Task 8 of 15) Variable references follow the hierarchical structure of blocks.

If the variable is defined in the current block, we use that declaration.

Otherwise, we look up the block in which the current block appears, and so on recursively. (Specifically, if the current block is a function body, the next block will be the block in which the function definition is; if the current block is the top-level block, the next block will be the **primordial block**.)

If the current block is already the primordial block and we still haven't found a corresponding declaration, the variable reference errors.

The primordial block is a non-visible block enclosing the top-level block. This block defines values and functions that are provided by the language itself.

# Scope

(Task 12 of 15) The **scope** of a variable is the region of a program where we can refer to the variable. Technically, it includes the block in which the variable is defined (including sub-blocks) *except* the sub-blocks where the same name is re-defined. When the exception happens, that is, when the same name is defined in a sub-block, we say that the variable in the sub-block **shadows** the variable in the outer block.

We say a variable reference (e.g., "the `x` in `x + 1`") is **in the scope of** a declaration (e.g., "the `x` in `x = 23`") if and only if the former refers to the latter.

(Task 13 of 15) Please select all statements that apply to the following program:

```
x = 45
def f():
    def g():
        return x + 1
    x = 67
    return x + 2
print(f())
print(x + 3)
```

- The scope of the top-level `x` includes the whole program.
- The scope of the top-level `x` includes `x + 3`.
- The top-level `x` shadows the `x` defined in `f`.
- The `x` in `x + 1` is in the scope of the `x` defined in `f`.
- The `x` in `x + 2` is in the scope of the `x` defined in `f`.
- The `x` in `x + 3` is in the scope of the `x` defined in `f`.

# Output ?

(Task 14 of 15) Here is a program that confused many students

Python [Run ▶]

```
x = 1
def foo():
    return x
def bar():
    x = 2
    return foo()
print(bar())
```

Lispy

```
(defvar x 1)
(deffun (foo)
  x)
(deffun (bar)
  (defvar x 2)
  (foo))

(bar)
```

Please

1. Run this program in the stacker by clicking one of the [Run ▶] buttons above;
2. The stacker would show how this program produces its result(s);
3. Keep clicking ⏭ Next until you reach a configuration that you find particularly helpful;
4. Click 🔗 Share This Configuration to get a link to your configuration;
5. Submit your link below;